Capítulo 3

Ejemplos

En este capítulo se presentarán diversos ejemplos implementados en los sistemas que abarcan el presente estudio. Los tipos de problemas fueron seleccionados para mostrar las técnicas de optimización soportadas y las funciones de agregación incluidas.

Cuando decimos que soporten funciones de agregación, nos referimos a aquellas operaciones aplicadas a conjuntos de elementos, adición, conteo, etc.
Para nosotros es importante que el problema sea resuelto más como un problema de optimización que de decisión. Un problema de optimización contesta la pregunta ¿cuál es la mejor solución? Mientras que un problema de decisión responde ¿existe una solución con una cierta característica? [R3.1]
Por ejemplo para el caso del problema de la "mochila binaria" [R3.2] se tiene el siguiente planteamiento:
Considera una mochila de capacidad *C*, donde *C* es un entero positivo. Sea *n* el número de objetos de tamaño (positivo) $s_1,s_2,...s_n$ cada uno con una ganancia de $p_1,p_2,...p_n$.
El problema de optimización consiste en encontrar la ganancia máxima de cualquier subconjunto de objetos que llenen la mochila, sin sobrepasar su capacidad.
El problema de decisión es dado un número *k*, ¿hay un subconjunto de objetos que llenen la mochila y cuya ganancia total sea al menos *k*? O de otra forma, ¿puede un valor de al menos *V,* ser alcanzado sin exceder el costo *C*?

Por otra parte, las versiones de los sistemas que fueron utilizadas para realizar las pruebas de los ejemplos de éste capítulo son :

- Smodels versión 2.27 y lparse versión 1.0.13
- A-POL versión 1.0c y Java Runtime Environment JRE 1.4.2
- DLV versión build BEN/Jun 26 2003 gcc 2.95.3-10 (cygwin special)
- Aset-Solver versiones:
  XSB 2.5 (Okocim) of March 11, 2002
  Setparse y aset versiones únicas construidas en 2001.

Para más información consultar el Apéndice A.

Finalmente para algunos de estos problemas se procedió a realizar un análisis comparativo entre ellos.

3.1 Mochila Binaria

[R1.1] La "mochila binaria" (0-1 *knapsack*) es un problema típico de programación dinámica. Éste consiste en llenar una mochila con varios objetos, cada objeto (*object*) tiene un peso (*weight*) y una ganancia (*profit or val*), el peso de los objetos escogidos no puede exceder la capacidad máxima de la mochila y la suma de sus ganancias debe ser óptima. La formulación del problema es la siguiente:

$$ks(n,k) = \text{máx}\begin{cases} 0 & , n = 0 \\ ks(n-1,k) & , 1 \leq n \\ profit(n) + ks(n-1, k - weight(n)) & , weight(n) \leq k \ y \ 1 \leq n \end{cases}$$

Donde *n* es el número de objetos y *k* es la capacidad de la mochila.
Con esta formulación no se recupera que objetos se seleccionan, solo la ganancia óptima. En A-POL usando la característica de "recuperación de testigo" es posible también obtener los objetos.

Este ejemplo fue implementado en A-POL, Aset-Solver, Smodels y DLV con la misma base de datos extensional.

Implementación en A-POL [R1.1]

```
%Enrique Corona
focus = take.

declare > ks(object,tcost).
declare > profit(object).
declare > sb(object,object).

select take1 where ks(3,10).

object(0..3).
tcost(0..10).

%Base de datos extensional

%tamaño o peso del objeto
cost(0,0).
cost(1,2).
cost(2,7).
cost(3,5).

%ganancia
profit(0)>= 0.
profit(1)>= 4.
profit(2)>= 3.
profit(3)>= 2.

%Reglas
ks(0,K)>= 0.

%cuando no se escoje un objeto en particular
ks(N,K)>= ks( sub(N,1), K ) :- le(1,N). / take1(N,K).

%cuando si se escoge un objeto en particular
ks(N,K)>= add( profit(N), ks( sub(N,1), sub(K,X) ) ) :- cost(N,X), le(X,K), le(1,N).
                                                        / take1(N,K).
```

```
% Recuperar que objetos se seleccionan
%take(X1,Y) :- take1(X1,Y), not take(X2,Y), X2 > X1, object(X2), tcost(Y).
%cambios D.M.O.
sb(N,M) >= sub(N,M) :- object(N), object(M).
take(1,Z) :- take1(1,Z), cost(1,R), Z >= R.
take(N,X) :- take1(N,X), sb(N,1)=M, take1(M,X1), tcost(X), tcost(X1),
                    object(N), object(M), X1 < X.
```

Salida:
> apol examples/knapsackConAset.pol
{take(1,3), take(2,10)}
Este resultado nos muestra que los objetos que se llevan son el 1 y 2 sin sobrepasar la capacidad de la mochila que es de 10.

Implementación en Smodels [R3.3]

```
%Ch. Baral
%http://www.baral.us/bookone/
%Index of  /bookone/code/smodels/temp/knapsack2.sm. 20-Mar-2003.

item(1..3).

%ganancia
weight val(1) = 4.
weight val(2) = 3.
weight val(3) = 2.


%tamaño del objeto
weight cost(1) = 2.
weight cost(2) = 7.
weight cost(3) = 5.

in_sack(X) :- item(X),  not not_in_sack(X).

not_in_sack(X) :- item(X), not in_sack(X).

val(X) :- item(X), in_sack(X).

cost(X) :- item(X), in_sack(X).

 cond1 :- [ cost(X) : item(X) ] 10.
 :- not cond1.

maximize [ val(X) : item(X) ].

hide item(X).
hide not_in_sack(X).
```

Salida:

> lparse knapsackConAs.sm|smodels 0
smodels version 2.27. Reading...done
Answer: 1
Stable Model: in_sack(1) in_sack(2) cond1
{ } min = 2
False
Duration: 0.000
Number of choice points: 2
Number of wrong choices: 2
Number of atoms: 19
Number of rules: 20
Number of picked atoms: 14
Number of forced atoms: 0
Number of truth assignments: 86
Size of searchspace (removed): 6 (0)

El modelo resultante es *Stable Model: in_sack(1) in_sack(2*), que incluye a los objetos 1 y 2 cuya ganancia máxima es $7. El resto de la salida es información estadística de la ejecución interna de smodels, tal como *False* que indica al usuario que no hay más modelos a ser calculados.
La implementación original en [R3.3] tenía un error dificil de detectar, la sentencia *maximize* estaba implementada como:
maximize ***{*** *val(X) : item(X)* ***}***
y lo correcto es
maximize ***[*** *val(X) : item(X)* ***]****.*
La primera sentencia genera un resultado incorrecto porque usando llaves {}, maximiza el número total de *items* y usando corchetes [] maximiza la ganancia, *val,* de los objetos, siendo esto último lo que se busca en el ejemplo.

Implementación en DLV [R3.3]

```
%Uriel Alejandro Mtz Huitle
% declaración de los objetos y de sus características
%nombre, val , costo
obj(a,4,2).
obj(b,3,7).
obj(c,2,5).

% definir la capacidad de mochila
capacidad(10).

% formación de los posibles conjuntos de objetos para introducir en la
%mochila
in_sack(X,V,C) :- obj(X,V,C),not not_in_sack(X,V,C).
not_in_sack(X,V,C) :- obj(X,V,C),not in_sack(X,V,C).

% midiendo que el valor total no sobre pase la capacidad de la mochila
s1 :- capacidad(K),#sum{C : in_sack(X,V,C)}<=K.
:- not s1.
```

% maximizar el número de elementos dentro de la mochila o
% minimizar el número de elementos que no están dentro de la mochila
:~ not_in_sack(X,V,C). [V:1]

Salida:
> dlv -nofacts knapsackConAset.dl
DLV [build BEN/Jun 26 2003 gcc 2.95.3-10 (cygwin special)]
*Best model: {**in_sack(a,4,2), in_sack(b,3,7),** not_in_sack(c,2,5), s1}*
Cost ([Weight:Level]): <[2:1]>

Esta salida muestra la versión de DLV usada, el modelo resultante con los objetos 1 y 2.

Observando los distintos códigos arriba mostrados, intuitivamente se puede afirmar que la implementación en DLV es de las más transparentes para entender, incluso partiendo de que es casi una copia de la implementación en Smodels; de hecho al programador se le dio como referencia el código en Smodels y la traducción a DLV fue casi directa, sin llegar a afirmar que sea la implementación óptima, podemos decir que al menos es la que se explica con más claridad.

3.2 FastFood

Este problema de optimización pertenece al archivo de problemas del *International Collegiate Programming Contest* de la ACM [R1.1] y fue uno de los problemas del concurso regional del suroeste de Europa en 1998. El enunciado original es el siguiente (traducido):
Una cadena de comida rápida tiene varios restaurantes a lo largo de una carretera y han decidido construir varios almacenes a lo largo de ella, cada almacén puede ser ubicado en un restaurante y suministrará a varios restaurantes con los ingredientes necesarios. Naturalmente estos almacenes deben ser colocados de tal forma que la distancia promedio entre un restaurante y su almacén designado se minimice. Usted debe escribir un programa que calcule las posiciones óptimas y las asignaciones de los almacenes.
Para precisar esto el gerente de la cadena ha establecido las siguientes especificaciones:

Se le darán las posiciones de *n* restaurantes a lo largo de la carretera donde $d_1 \leq d_2 \leq \ldots \leq d_n$ (estas son las distancias medidas desde las oficinas principales de la compañía, que se encuentra en la misma carretera).
También se da un número $k \leq n$ como el número de almacenes a construir.
Los k almacenes se construirán en las ubicaciones de k diferentes restaurantes. Cada restaurante será asignado al almacén más cercano, del que será abastecido. Para minimizar los costos de envío, la suma de la distancia total definida por:

$$\sum_{i=1}^{n} \left| d_i - \quad (\textit{posición del almacén sirviendo al restaurante } i) \right|$$

Debe ser lo menor posible. Escriba un programa que calcule las posiciones de los k almacenes, de forma que la suma de la distancia total sea minimizada.

Este ejemplo fué implementado en A-POL y DLV con la misma base de datos extensional

Implementación en A-POL [R1.1]

```
%Enrique Corona
focus = location.

declare < middle(dom,dom).
declare < rest(dom).
declare < sumato(dom,dom,dom).
declare < cost(dom,dom).
declare < best(dom,dom).
declare < rdist(dom,dom).


%restaurantDLV.in número de restaurante y ubicación
%1 5
%2 6
%3 12
%4 19
%5 20
%6 27

dom(1..6).
fill rest with examples/fastfood/restaurantDLV.
select location where best(3,6).

middle(X,Y)<= div(add(X,Y),2).
rdist(I,J)<= abs(rest(I),rest(J)).
cost(I,J)<= sumato(I,J,middle(I,J)).

sumato(I,I,K)<= rdist(I,K).
sumato(I,J,K)<= add( rdist(I,K), sumato(add(I,1),J,K) ) :- lt(I,J).

best(I,I)<= 0. / location(I).
best(1,I)<= cost(1,I) :- middle(1,I) = M. / location(M).
best(I,J)<= add( cost(B,J), best( sub(I,1), sub(B,1) ) ) :- middle(B,J) = M, lt(I,J), le(I,B),
le(B,J), dom(B). / location(M).
```

Salida:
> apol examples/fastfood/fastfood.pol
{location(2), location(4), location(6)}

El cambio en los datos para adecuarlos a los mismos que en DLV, forzó a modificar en A-POL varios archivos.

Implementación en DLV [R1.1]

```
%Enrique Corona
#maxint=27.
```

```
rest(1, 5).  rest(2, 6).  rest(3, 12).  rest(4, 19).  rest(5, 20).  rest(6, 27).
ndepot(3).
nrest(6).

dom(1..6).

% reglas
abs(X,Y,R) :- X = R+Y, X>=Y, #int(X), #int(Y), #int(R).
abs(X,Y,R) :- Y = R+X, X<Y, #int(X), #int(Y), #int(R).

middle(X,X,X) :- #int(X).
middle(X,Y,X) :- abs(X,Y,1).
middle(X,Y,R) :- middle(R1,R2,R), R1 = X+1, Y = R2+1, R1<=R2.

rdist(I,J,R) :- abs(R1,R2,R), rest(I,R1), rest(J,R2).

sumato(I,I,K,R) :- rdist(I,K,R).
sumato(I,J,K,R) :- R = R1+R2, rdist(I,K,R1), sumato(R3,J,K,R2), R3 = I+1, I<J.

cost(I,J,R) :- sumato(I,J,K,R), middle(I,J,K).

best1(I,I,0) :- dom(I).
best1(1,I,R) :- cost(1,I,R).
best1(I,J,R) :- R = R1+R2, cost(B,J,R1), best1(R3,R4,R2),
                I = R3+1, B = R4+1, I<J, I<=B, B<=J.

best2(I,J,R) :- best1(I,J,R), not rbest(I,J,R).
rbest(I,J,R) :- best1(I,J,M), #int(R), M<R.
best(I,J,R) :- best2(I,J,R), ndepot(I), nrest(J).
```

Salida:

```
>dlv -filter=best fastfood.dl
DLV [build BEN/Jun 26 2003   gcc 2.95.3-10 (cygwin special)]
{best(3,6,8)}
```

Las dos implementaciones de este problema lucen demasiado recargadas de llamadas a funciones.
La salida de A-POL muestra donde se colocarán los depósitos y DLV muestra el valor total de la distancia más corta entre todos los restaurantes y sus depósitos.
Se tiene este ejemplo implementado por Nicola Leone de DLV, pero desafortunadamente usa una variante de la función de agregación *#min* que aún es experimental en la versión liberada del sistema, por lo que no trabaja apropiadamente, generando modelos vacíos.

3.3 Camino más corto entre dos nodos de un grafo

Supongamos que un grafo está definido por predicados del tipo *edge(X,Y,C) o arco(X,Y,C),* donde *C* es la distancia positiva entre *X* y *Y*, el problema es calcular la distancia más corta entre dos nodos dados. Este problema tiene una variedad de

aplicaciones por ejemplo en electrónica, comunicaciones o transporte donde es importante encontrar la mejor ruta entre dos ciudades dadas.

Implementación en A-POL [R1.1]

```
%Enrique Corona
%dominio que define a la función sh
declare < sh(node,node).

%indica que testigo se recupera y muestra el camino más corto desde a hasta c
select way where sh(X,Y)<- start(X), end(Y).

%base de datos extensional
node(a).
node(b).
node(c).
node(d).

edge(a,b,1).
edge(b,c,2).
edge(a,d,4).
edge(d,c,1).

start(a).
end(c).

%la siguiente cláusula de orden parcial obtiene la distancia por un camino directo
sh(X,Y) <= C :- edge(X,Y,C)./ way(X,Y).

%obtiene la distancia por caminos alternativos, pasando por otros nodos
%entre X y Y.
sh(X,Y) <= add(sh(X,Z),sh(Z,Y)) :- edge(X,Z,C).
```

Salida:
>apol examples/short2.pol
{ sh(a,b,1), ***sh(a,c,3),*** *sh(a,d,4), sh(b,c,2), sh(d,c,1),* ***way(a,b), way(b,c)****}*

La salida nos muestra el peso del camino más corto como *sh(a,c,3)* y también observamos la característica de A-POL llamada "recuperación de testigo" que nos muestra la secuencia de los nodos como *way(a,b), way(b,c).*

Implementación en DLV

```
%Pedro Alejandro Miranda V.   #is117282
#maxint=10.
%base extensional
c(a).
c(b).
c(c).
c(d).
```

arco(a,b,1).
arco(b,c,2).
arco(a,d,4).
arco(d,c,1).

inicio(a).
fin(c).

%base intensional
minimo(W) :- #min{Z : camino(X,Y,Z)} = W, inicio(X), fin(Y).

camino(X,Y,Z):-arco(X,Y,Z).
camino(X,Y,Z):-camino(X,W,M), camino(W,Y,N), X!=Y, Z=M+N.

Salida:
> dlv -nofacts MasCorto.dl
DLV [build BEN/May 16 2003 gcc 2.95.3 20010315 (release)]
{minimo(3), camino(a,b,1), camino(a,c,3), *camino(a,c,5),camino(a,d,4),* ***camino(b,c,2),*** *camino(d,c,1)}*

En este ejemplo también nos muestra el camino desde *a* a *c,* pero también incluye otros caminos y para el caso que sean muchos nodos, la salida mostraría todos los caminos y muchas veces es dificil distinguir el que buscamos.

3.4 La compañía Cartoon Co.

El siguiente ejemplo fue propuesto en el manual de usuario de DLV [R1.6] y muestra el uso y sintáxis de las funciones de agregación.
Los datos representan a los empleados de una compañía llamada *Cartoons Co.* con su nombre, número y salario. Consiste en obtener:
Cuantos empleados de la compañía ganan mas de 1000.
Cuanto gasta la compañía en salarios.
Cual es salario mínimo pagado.
Cual es salario máximo pagado.
Cada empleado está representado por un hecho de la forma *emp(id,name,salary)*

Implementación en A-POL [R1.1]
%Enrique Corona

declare < emp(id,name).
declare > over1000(id).
declare < min(none).
declare > max(none).
declare > over1000aux(dom).
declare > over1000nr(none).
declare > salaryTotalAux(dom).
declare > salaryTotal(none).
none(void).

```
dom(0..10).

%Base de datos extensional
id(1). id(2). id(3). id(4). id(5).

name(goofie). name(willy). name(woody). name(jerry). name(tom).

emp(1,goofie,1250).     emp(2,willy,700).
emp(3,woody,750).      emp(4,jerry,900).
emp(5,tom,1050).

%reglas
%Cual es el empleado que gana más y cual gana menos
over1000(I)>= S :- emp(I,E,S), ge(S,1000).

min(void)<= S :- emp(I,E,S). / poor(E,S).
max(void)>= S :- emp(I,E,S). / rich(E,S).
select poor where min(void).
select rich where max(void).

over1000aux(I)>= over1000aux(sub(I,1)).
over1000aux(I)>= add( over1000aux(sub(I,1)), bool(over1000(I)) ).
```

over1000nr(void)>= over1000aux(I) :- id(I).

```
salaryTotalAux(I)>= salaryTotalAux(sub(I,1)).

salaryTotalAux(I)>= add( salaryTotalAux( sub(I,1) ), S ) :- emp(I,E,S).
```

salaryTotal(void)>= salaryTotalAux(I) :- id(I).

Salida:
> apol examples/cartoon.pol
{ max(void,1250), min(void,700), over1000(1,1250), over1000(5,1050), poor(willy,700), rich(goofie,1250)}

Implementación en DLV[R1.6]

```
%
emp(1,goofie,1250).    emp(2,willy,700).
emp(3,woody,750).     emp(4,jerry,900).
emp(5,tom,1050).

%Cuantos empleados de la compañía ganan mas de 1000.
%Aquí se usa  #count que regresa la cardinalidad de un conjunto al cual se aplica.
over1000(I,N) :- emp(I,N,S), S > 1000.
over1000nr(X) :- #count{I : over1000(I,N)} = X.

%Avisar si un empleado gana mas de 1200
warnMeOver1200 :- #count{I : emp(I,N,S), S > 1200} > 0.
```

```
%Se desea saber cuanto gasta la compañía en salarios.
%Aquí se aplica #sum
salaryTotal(X) :- #sum{S : emp(I,N,S)} = X.

%Avisar si los gastos por salario exceden a 4500
warnSalUp :- #sum{S : emp(I,N,S)} > 4500.

%Cual es el salario mínimo pagado entre todos los empleados.
%Aquí se usa  #min que regresa el minimo valor de un conjunto
 lowest(X) :- #min  {S : emp(I,N,S)} = X.

%Cual es el salario máximo pagado.
%#max regresa el maximo valor de la variable dada en el conjunto de simbolos.
highest(X) :- #max{S : emp(I,N,S)} = X.
```

Salida:

```
 >dlv -nofacts cartoonco.dl
DLV [build BEN/Jun 26 2003   gcc 2.95.3-10 (cygwin special)]

{ over1000(1,goofie), over1000(5,tom),
  over1000nr(2),
  warnMeOver1200,
  salaryTotal(4650),
  warnSalUp,
  lowest(700),
  highest(1250)   }
```

Por alguna razón en la versión más reciente de A-POL no muestra la salida completa, aquella que incluya tanto el número de empleados que ganan más de 1000 y cuanto gasta la compañía en salarios. No obstante el mismo código ejecutado en la versión anterior del sistema (*apol-1.0b*) si genera el resultado correcto.
Podemos observar que el uso de las funciones de agregación en DLV es bastante intuitivo.

3.5 Un problema de combinatoria, subasta de objetos.

[R3.3] Un licitador tiene permitido hacer una oferta por un conjunto de objetos que le interesen. El subastador deberá seleccionar un subconjunto de tales objetos de tal forma que obtenga el precio máximo por ellos, asegurándose que no aceptará ofertas que contengan el mismo elemento, ya que cada objeto puede venderse solamente una vez. El número de objetos en total es 4, {1,2,3,4} y las propuestas de los compradores son {a,b,c,d,e} donde a=[{1,2,3},$24], esto significa que el comprador *a* esta interesado en los objetos 1,2,3 a un costo de $24. De la misma forma para el resto. En resumen el problema es maximizar la ganancia total obtenida, aceptando el mejor subconjunto de ofertas donde ninguna de ellas contengan el mismo objeto.

Implementación en DLV [R3.3]

```
% PhD. Chitta Baral
%http://www.baral.us/bookone/
```

```
% Entrar en la liga /code/dlv/auctions

bid(a).    bid(b).    bid(c).    bid(d).    bid(e).

 in(1,a).   in(2,a).   in(3,a).

 in(2,b).   in(3,b).

 in(3,c).   in(4,c).

 in(2,d).   in(3,d).   in(4,d).

 in(1,e).   in(4,e).

%Reglas
  sel(X) v not_sel(X) :- bid(X).

:- bid(X),
    bid(Y),
    sel(X),
    sel(Y),
    X != Y,
    in(I,X),
    in(I,Y).

 :~ not sel(a). [24:1]
 :~ not sel(b). [9:1]
 :~ not sel(c). [8:1]
 :~ not sel(d). [25:1]
 :~ not sel(e). [15:1]
```

Salida:
> dlv -nofacts bidders.dl
DLV [build BEN/May 16 2003   gcc 2.95.3 20010315 (release)]
Best model: {not_sel(a), not_sel(b), not_sel(c), ***sel(d),*** *not_sel(e)}*
Cost ([Weight:Level]): <[56:1]>

Se emplearon *weak constraints* para obtener el valor óptimo.

Implementación en Smodels [R3.3]

```
%Ch. Baral, page 402
%http://www.baral.us/bookone/
%Index of /code/smodels/comb-auc

item(1..4).
bid(a;b;c;d;e).
in(1,a).   in(2,a).   in(3,a).

in(2,b).   in(3,b).
```

```
in(3,c).   in(4,c).

in(2,d).   in(3,d).   in(4,d).

in(1,e).   in(4,e).

weight sel(a) = 24.
weight sel(b) = 9.
weight sel(c) = 8.
weight sel(d) = 25.
weight sel(e) = 15.

sel(X) :- bid(X), not not_sel(X).
not_sel(X) :- bid(X), not sel(X).

%two different bids with the same items can not be selected.
:- bid(X), bid(Y), sel(X), sel(Y), not eq(X,Y), item(I), in(I,X), in(I,Y).

%Select the bids such that their total weight is maximized.
maximize [sel(X) : bid(X)].

hide bid(X).
hide not_sel(X).
hide item(X).
hide in(X,Y).
```

Salida 1:

```
> lparse -d none bidders.sm| smodels 0
smodels version 2.27. Reading...done
Answer: 1
Stable Model: sel(e) sel(b)
{ not sel(e), not sel(d), not sel(c), not sel(b), not sel(a) } min = 57
Answer: 2
Stable Model: sel(d)
{ not sel(e), not sel(d), not sel(c), not sel(b), not sel(a) } min = 56
False
Duration: 0.010
```

Salida 2:

```
 > lparse bidders.sm| smodels 0
smodels version 2.27. Reading...done
Answer: 1
Stable Model: sel(a)
{ not sel(e), not sel(d), not sel(c), not sel(b), not sel(a) } min = 57
Answer: 2
Stable Model: sel(d)
{ not sel(e), not sel(d), not sel(c), not sel(b), not sel(a) } min = 56
False
Duration: 0.000
```

Salida 3:
> lparse -d all bidders.sm|smodels 0
smodels version 2.27. Reading...done
Answer: 1
Stable Model: sel(d)
{ not sel(e), not sel(d), not sel(c), not sel(b), not sel(a) } min = 56
False
Duration: 0.010

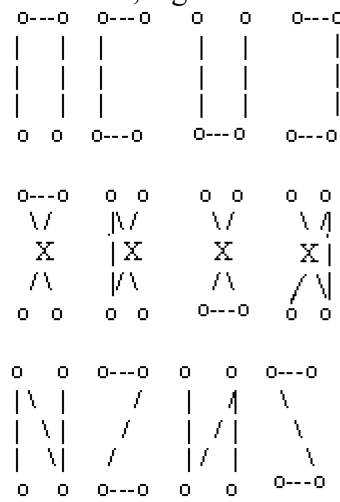
Aquí mostramos que usando diferentes opciones para *lparse*, *smodels* muestra diferentes modelos resultantes. Este comportamiento no es entendible, ni aunque los diseñadores explican que como el programa usa *maximize,* ésta sentencia provoca la alteración del orden de salida de los modelos, pero el último modelo siempre es el óptimo.

3.6 Mínimo árbol abarcador

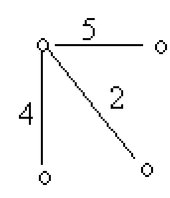
[R3.4] Un árbol abarcador de un grafo es un subgrafo que contiene a todos los vértices del grafo. Un mismo grafo puede contener muchos árboles abarcadores. Por ejemplo un grafo de 4 vértices se ve así:

```
o---o
| \/ |
| X |
| /\ |
o---o
```

Este grafo tiene 16 árboles abarcadores, algunos son:



El camino entre los vértices del grafo tiene un valor, puede llamarse peso o longitud.



El peso del árbol es la suma del peso de sus vértices. Para el caso anterior el peso total del árbol es 11.
Obviamente diferentes árboles tienen diferentes longitudes.
El problema es encontrar el mínimo árbol abarcador, esto es el árbol con menor longitud o peso y que cumpla con que todos los nodos son alcanzados bajo las siguientes condiciones:
La raíz del árbol no debe tener ningún arco de entrada
Cada nodo en el árbol debe tener solo un arco de entrada.

Implementación en DLV [R1.6]

```
% http://www.dbai.tuwien.ac.at/proj/dlv/man/#MIN-SPANNING-TREE
% Usando weak constraints
root(a).
node(a). node(b). node(c). node(d). node(e).
edge(a,b,4). edge(a,c,3). edge(c,b,2). edge(c,d,3). edge(b,e,4).
edge(d,e,5).

in_tree(X,Y,C) v out_tree(X,Y) :- edge(X,Y,C), reached(X).

%La raíz del árbol no debe tener ningún arco de entrada
:- root(X), in_tree(_,X,C).

%Cada nodo en el árbol debe tener solamente un arco de entrada
:- in_tree(X,Y,C), in_tree(Z,Y,C), X != Z.

%Cada nodo en el grafo debe ser alcanzado, esto es que pertenezca al árbol
%abarcador
reached(X):- root(X).
reached(Y):- reached(X), in_tree(X,Y,C).
:- node(X), not reached(X).

:~ in_tree(X,Y,C). [C:1]
```

Salida:
> dlv -nofacts min_sp.txt
DLV [build BEN/May 16 2003   gcc 2.95.3 20010315 (release)]

Best model: { reached(a), out_tree(a,b), ***in_tree(a,c,3),*** *reached(b), reached(c),* ***in_tree(b,e,4), in_tree(c,b,2), in_tree(c,d,3),*** *reached(e), reached(d), out_tree(d,e)}*
Cost ([Weight:Level]): <[12:1]>

El mismo ejemplo se implementa de nuevo, pero ahora usando funciones de agregación. Según los autores, es una forma más elegante de resolverlo, sin usar reglas auxiliares, y mejora la lectura y el entendimiento del problema, ya que las funciones de agregación son más claras al usuario.

Implementación en DLV, usando funciones de agregación[R1.6]

```
root(a).

node(a). node(b). node(c). node(d). node(e).

edge(a,b,4). edge(a,c,3). edge(c,b,2). edge(c,d,3). edge(b,e,4). edge(d,e,5).

in_tree(X,Y,C) v out_tree(X,Y) :- edge(X,Y,C).

%La raiz del árbol no debe tener ningun arco de entrada
:- root(R), not #count{X : in_tree(X,R,C)} = 0.

%Cada nodo en el árbol debe tener solamente un arco de entrada
%esto esta mal porque contempla a todos los nodos por igual incluso la raíz
%:- edge(_,Y,C), not #count{X : in_tree(X,Y,C)} = 1.

%From Nicola Leone´s mail: should ensure that each non-root node has
% precisely 1 incoming arc.
%Cada nodo en el grafo debe ser alcanzado, es decir asegurar que pertenece
% al árbol .
 :- node(Y), not root(Y), not #count{X : in_tree(X,Y,C)} = 1.

:~ in_tree(X,Y,C). [C:1]
```

Salida:
> dlv -nofacts min_spWITHAggregateFnc.dl
DLV [build BEN/May 16 2003   gcc 2.95.3 20010315 (release)]

Best model: { out_tree(a,b), ***in_tree(a,c,3), in_tree(b,e,4), in_tree(c,b,2), in_tree(c,d,3)****, out_tree(d,e)}*
Cost ([Weight:Level]): <[12:1]>

Este ejemplo fue tomado del tutorial de DLV, pero en un primer intento, no generó la salida correcta, el error fue remitido a los diseñadores del sistema quienes modificaron el código.

Implementación en Smodels

La siguiente implementación fue realizada apoyándose en el código anterior de DLV, como se puede observar los cambios solamente se refieren a la sustitución de las cláusulas disyuntivas y el uso de *maximize* en lugar de *weak constraints*, el resto del código es muy semejante.

```
%Silvia del Carmen Serrano Ramos #118602
%Miguel Angel Meza Morales        #118749
%Juan Islas Aparicio              #119100
%16 de marzo del 2004

 root(a).  node(a).   node(b).   node(c).   node(d).   node(e).
```

```
edge(a,b,4).   edge(a,c,3).    edge(c,b,2).    edge(c,d,3).    edge(b,e,4).  edge(d,e,5).

weight edge1(a,b)=4.
weight edge1(a,c)=3.
weight edge1(c,b)=2.
weight edge1(c,d)=3.
weight edge1(b,e)=4.
weight edge1(d,e)=5.

edge1(X,Y):-in_tree(X,Y,C),edge(X,Y,C).

in_tree1(X,Y):-in_tree(X,Y,C),edge(X,Y,C).
in_tree(X,Y,C):-edge(X,Y,C), reached(X),not not_in_tree(X,Y,C).
not_in_tree(X,Y,C):- edge(X,Y,C),reached(X), not in_tree(X,Y,C).

:- root(X), in_tree(W,X,C),edge(W,X,C).
:- in_tree(X,Y,C), in_tree(Z,Y,C), X != Z,edge(X,Y,C),edge(Z,Y,C).

reached(X):- root(X).
reached(Y):- reached(X), in_tree(X,Y,C),edge(X,Y,C).
:- node(X), not reached(X).

minimize{edge1(X,Y):in_tree1(X,Y)}.


hide edge(X,Y,C).
hide in_tree(X,Y,C).
hide not_in_tree(X,Y,C).
hide reached(X).
hide root(X).
hide node(X).
```

Salida:
> lparse -d none arbolabarcadorAlumnos.sm|smodels 0
smodels version 2.27. Reading...45: A non-domain condition predicate 'edge1'
done
Answer: 1
Stable Model: edge1(b,e) edge1(c,d) edge1(c,b) edge1(a,c)
in_tree1(b,e) in_tree1(c,d) in_tree1(c,b) in_tree1(a,c)
{ } min = 0
False
Duration: 0.000

3.7 Formación de equipos

[R1.6] Un equipo de trabajo debe ser formado de un conjunto de empleados de acuerdo a las siguientes características:
R1. El equipo constará solo de un número límite de empleados

R2. Un equipo requerirá de un mínimo de habilidades diferentes.
R3. La suma de los salarios de los miembros del equipo no excederá el presupuesto dado.
R4. El salario de cada empleado tiene un margen especificado
R5. El número de mujeres en el equipo tiene que alcanzar un número dado.

Este ejercicio ejemplifica el uso de funciones de agregación en DLV. Se observa que éstas funciones son usadas transparentemente y traducen directamente los requerimientos solicitados.

Implementación en DLV [R1.6]

```
employee(10,m,990,1000).
employee(20,f,990,300).
employee(30,m,980,400).
employee(40,f,980,2000).
employee(50,f,940,600).
employee(60,m,940,600).

 %the size of the team
 nEmp(3).

 %the minimum number of different skills
 nSkill(3).

 maxSal(1500).

 %minimum number of women
 minwomen(2).

 %el presupuesto
 budget(2000).

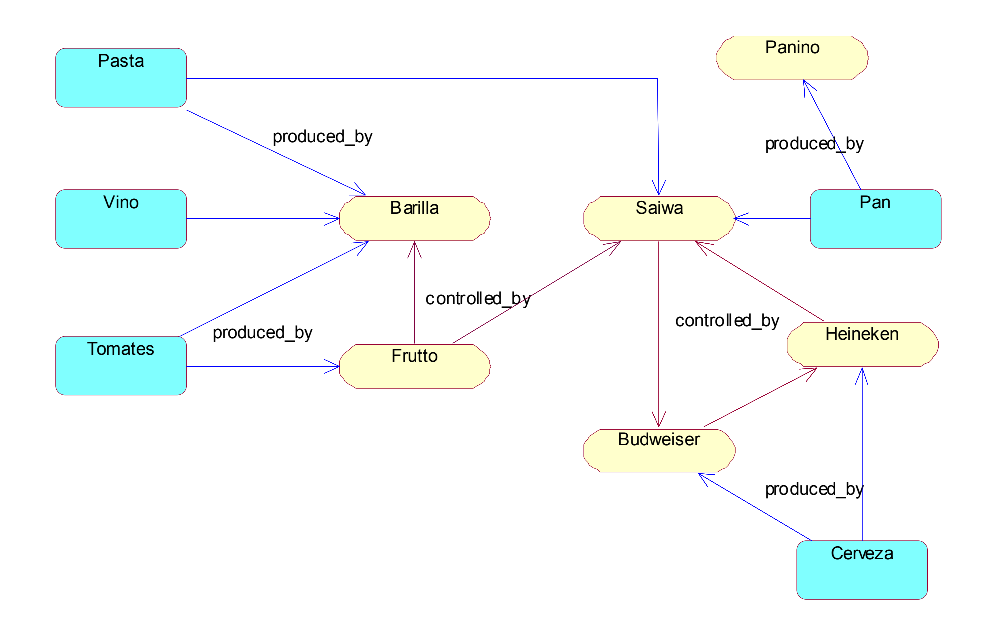
 %Reglas
 %un empleado es incluido en el equipo o no
 in(Id) v out(Id) :-  employee(Id,Sex,Skill,Salary).


 %cubre las restricciones r1 - r5
:- nEmp(N),     not #count{Id          : in(Id)} = N.
:- nSkill(M),     not #count{Skill      : employee(Id,Sex,Skill,Salary),in(Id)} >= M.
:- budget(B),    not  #sum{Salary,Id : employee(Id,Sex,Skill,Salary),in(Id)}<= B.
:- maxSal(M),   not #max{Salary     : employee(Id,Sex,Skill,Salary),in(Id)}<= M.
:- minwomen(W),not #count{Id       : employee(Id,f,Skill,Salary),in(Id)} >= W.
```

Salida:
>dlv -nofacts teambuilding.dl
DLV [build BEN/Jun 26 2003   gcc 2.95.3-10 (cygwin special)]
{out(10), in(20), in(30), out(40), in(50), out(60)}

3.8 Compañías estratégicas

Este problema es referenciado como complejo [R2.9] y definido en [R2.10]. Se incluyó porque como se menciona en el capítulo 2, el equipo de DLV asegura que no se puede representar en Smodels.
Consiste en que existen ciertas compañías que controlan otras compañías, cada una produce un conjunto de productos y cada uno de ellos puede ser producido a lo más por dos compañías. Nosotros tenemos que vender algunas compañías, sin dejar de producir algún producto, por lo que se busca permanecer con el mínimo número de ellas. A las compañías que pertenecen en al menos uno de tales conjuntos se les denomina estratégicas. También se le califica a una compañía como estratégica si es controlada por otras compañías estratégicas. Por ejemplo:



Implementación en DLV

```
% http://www.dbai.tuwien.ac.at/proj/dlv/
 produced_by(pasta   , barilla ,saiwa).
 produced_by(tomatoes, frutto  ,barilla).
 produced_by(wine    , barilla ,0).
 produced_by(bread   , saiwa   ,panino).
 produced_by(beer    , heineken,budweiser).

 controlled_by(frutto   , barilla  ,saiwa,0).
 controlled_by(budweiser, heineken ,0    ,0).
 controlled_by(heineken , saiwa    ,0    ,0).
 controlled_by(saiwa    , budweiser,0    ,0).

% As we want to produce X, Y or Z must be strategic.
strategic(Y) v strategic(Z) :- produced_by(X,Y,Z).

% W is strategic, if it is controlled by strategic
% companies X, Y, and Z
```

```
strategic(W) :- controlled_by(W,X,Y,Z),
             strategic(X), strategic(Y), strategic(Z).
:- strategic(0).
```

Salida:
>dlv -nofacts strategic05.dl
DLV [build BEN/Jun 26 2003   gcc 2.95.3-10 (cygwin special)]
{strategic(barilla), strategic(panino), strategic(budweiser)}
{strategic(barilla), strategic(saiwa), strategic(budweiser)}
{strategic(barilla), strategic(panino), strategic(heineken)}
{strategic(barilla), strategic(saiwa), strategic(heineken)}

Implementación en Smodels

```
produced_by(pasta   , barilla ,saiwa).
 produced_by(tomatoes, frutto  ,barilla).
 produced_by(wine    , barilla ,0).
 produced_by(bread   , saiwa   ,panino).
 produced_by(beer    , heineken,budweiser).

 controlled_by(frutto   , barilla  ,saiwa,0).
 controlled_by(budweiser, heineken ,0    ,0).
 controlled_by(heineken , saiwa    ,0    ,0).
 controlled_by(saiwa    , budweiser,0    ,0).


{strategic(Y), strategic(Z)} :- produced_by(X,Y,Z).
strategic(W) :- controlled_by(W,X,Y,Z),
                   strategic(X), strategic(Y), strategic(Z).
:- strategic(0).

hide produced_by(X,Y,Z).
hide controlled_by(W,X,Y,Z).
```

Salida:
> lparse -d none strategic05.sm| smodels 0
smodels version 2.27. Reading...done
Answer: 1
Stable Model:
Answer: 2
Stable Model: strategic(budweiser)
Answer: 3
Stable Model: strategic(panino)
Answer: 4
Stable Model: strategic(budweiser) strategic(panino)
Answer: 5
Stable Model: strategic(heineken)
Answer: 6
Stable Model: strategic(heineken) strategic(panino)
Answer: 7

Stable Model: strategic(budweiser) strategic(heineken)
Answer: 8
Stable Model: strategic(budweiser) strategic(heineken) strategic(panino)
Answer: 9
Stable Model: strategic(frutto)
Answer: 10
Stable Model: strategic(budweiser) strategic(frutto)
........MAS COMBINACIONES DE MODELOS
Answer: 63
Stable Model: strategic(saiwa) strategic(budweiser) strategic(heineken) strategic(frutto) strategic(panino)
Answer: 64
Stable Model: strategic(saiwa) strategic(budweiser) strategic(heineken) strategic(frutto) strategic(barilla) strategic(panino)
False

Para codificar este problema en Smodels, cuyo planteamiento obliga a representarlo como disyuntivo, se aplicó la conversión sugerida en su manual, pero, como se puede observar, no generó los modelos correctos, por lo que comprobamos la afirmación hecha por el equipo de DLV, de que Smodels aun no soporta programación lógica disyuntiva.

3.9 Otros problemas

Existen en la literatura diversos problemas de optimización, que aunque no pudieron ser incluidos en el presente trabajo, es importante mencionarlos.
En [R3.5] se resuelven varios problemas que no son triviales, como el encontrar una secuencia que minimice el tiempo en el que un grupo de músicos pertenecientes a una orquesta, asistan a una sesión de ensayo, pero con espacios en donde no tocan sus instrumentos, por lo que lo ideal sería que llegaran en la primera pieza que intervienen y se vayan enseguida, de tal forma que sus tiempos de espera sean los mínimos, aquí se muestra como adaptar un problema con restricciones a un problema de optimización, otra variante es planear la filmación de una película de tal forma que optimice los costos por mantener desocupado a los actores, entre una escena y otra.
En [R3.3] se pueden consultar varios ejercicios en DLV y Smodels que ejemplifican el uso de las funciones de agregación. Entre estos se encuentra C*omputeagregation* escrito en Smodels.
En general es posible que casi cualquier problema pueda ser convertido a su variante de optimización, por ejemplo el problema de "*Graph Colouring*", se plantea como el colorear el grafo con un mínimo número de colores, mientras que la variante de decisión es la más común, decidir para un número particular de colores, si es posible colorear el grafo, ambos cumpliendo con la restricción de que 2 nodos adyacentes no pueden tener el mismo color. Más de estos pueden ser encontrados en [R3.6]
Para DLV otros problemas interesantes son "Seating Problem" y "Traveling Sales Person" disponibles en el manual de usuario.

3.10 Observaciones generales

Analizando los ejemplos anteriores podemos hacer algunas observaciones generales:

A-POL muestra ser un sistema que permite fácilmente representar un problema propuesto en principios matemáticos, pero por otra parte requiere tener el suficiente conocimiento del lenguaje para hacer uso de esa expresividad.

Analizando los problemas 3.1 y 3.6, observamos que traducir un código de Smodels a DLV y viceversa es bastante intuitivo y no representó cambios complejos o difíciles al programador, por lo que se puede vislumbrar que el paso de un sistema a otro puede efectuarse sin mucha complejidad.

Para nuestra sorpresa todos los sistemas, con excepción de Aset-Solver, mostraron deficiencias, incluso con ejemplos, que en su mayoría, ya habían sido probados por los diseñadores.
El caso de Aset-Solver es separado porque no tiene ejemplos que cumplieran con nuestras expectativas, por lo que no hubo suficiente información para evaluarlo correctamente.

Resumen del capítulo

En este capítulo se presentaron varios ejemplos probados en los diferentes sistemas; para algunos de ellos también se realizaron pruebas comparativas. A continuación se explican los resultados y conclusiones de este trabajo.